Parity Models: Erasure-Coded Resilience for Prediction Serving Systems

Jack Kosaian
Carnegie Mellon University
jkosaian@cs.cmu.edu

K. V. Rashmi
Carnegie Mellon University
rvinayak@cs.cmu.edu

Shivaram Venkataraman
University of Wisconsin-Madison
shivaram@cs.wisc.edu

Abstract

Machine learning models are becoming the primary workhorses for many applications. Services deploy models through prediction serving systems that take in queries and return predictions by performing inference on models. Prediction serving systems are commonly run on many machines in cluster settings, and thus are prone to slowdowns and failures that inflate tail latency. Erasure coding is a popular technique for achieving resource-efficient resilience to data unavailability in storage and communication systems. However, existing approaches for imparting erasure-coded resilience to distributed computation apply only to a severely limited class of functions, precluding their use for many serving workloads, such as neural network inference.

We introduce *parity models*, a new approach for enabling erasure-coded resilience in prediction serving systems. A parity model is a neural network trained to transform erasure-coded queries into a form that enables a decoder to reconstruct slow or failed predictions. We implement parity models in ParM, a prediction serving system that makes use of erasure-coded resilience. ParM encodes multiple queries into a "parity query," performs inference over parity queries using parity models, and decodes approximations of unavailable predictions by using the output of a parity model. We showcase the applicability of parity models to image classification, speech recognition, and object localization tasks. Using parity models, ParM reduces the gap between 99.9th percentile and median latency by up to 3.5×, while maintaining the same median. These results display the potential of parity models to unlock a new avenue to imparting resource-efficient resilience to prediction serving systems.

Jack Kosaian is supported by an SOSP 2019 student scholarship from the ACM Special Interest Group in Operating Systems.

SOSP '19, October 27–30, 2019, Huntsville, ON, Canada

ACM ISBN 978-1-4503-6873-5/19/10…$15.00
https://doi.org/10.1145/3341301.3359654

CCS Concepts • **Computer systems organization** → **Redundancy**; *Reliability*.

Keywords erasure coding, machine learning, inference

ACM Reference Format:
Jack Kosaian, K. V. Rashmi, and Shivaram Venkataraman. 2019. Parity Models: Erasure-Coded Resilience for Prediction Serving Systems. In *ACM SIGOPS 27th Symposium on Operating Systems Principles (SOSP '19), October 27–30, 2019, Huntsville, ON, Canada.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3341301.3359654

1 Introduction

Machine learning is widely deployed in production services and user-facing applications [1, 5, 7, 15, 21]. This has increased the importance of inference, the process of returning a prediction from a trained machine learning model. *Prediction serving systems* are platforms that host models for inference and deliver predictions for input queries. Numerous prediction serving systems are being developed by service providers [3, 4, 8] and open-source communities [9, 25, 64].

To meet the demands of user-facing production services, prediction serving systems must deliver predictions with low latency (e.g., within tens of milliseconds [25]). Similar to other latency-sensitive services, prediction services must adhere to strict service level objectives (SLOs). Queries that are not completed by their SLO are often useless to applications [15]. In order to reduce SLO violations, prediction serving systems must minimize tail latency.

Prediction serving systems often employ distributed architectures to support high query rate [25]. As depicted in Figure 1 (ignoring the purple components for the moment), a prediction serving system consists of a frontend which receives queries and dispatches them to one or more model instances. Model instances perform inference and return predictions. This distributed setup is typically run in large-scale, multi-tenant clusters (e.g., public clouds), where tail latency inflation is a common problem [26]. There are numerous causes of inflated tail latencies in these settings, such as multi-tenancy and resource contention [38, 44, 88], hardware unreliability and failures [19], and other complex runtime interactions [18]. Within the context of prediction serving systems, network and computation contention have both been shown as potential bottlenecks [25, 38], and routines like loading a new model can also cause latency spikes [64].

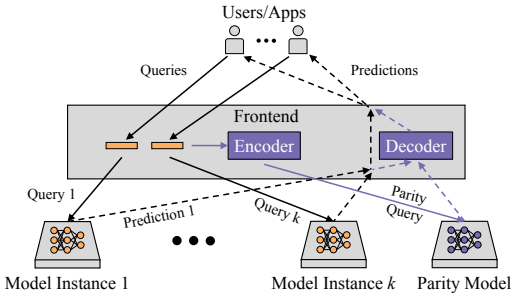


Figure 1. Architecture of a prediction serving system along with components introduced by ParM (shown in purple).

Due to the many causes of tail latency inflation, it is important for mitigations to be agnostic to the cause of slowdown [26]. However, current agnostic approaches for mitigating tail latency inflation either require significant resource overhead by replicating queries, or sacrifice latency by waiting to detect a slowdown or failure before retrying.

Erasure codes are popular tools for imparting resilience to data unavailability while remaining agnostic to the cause of unavailability and using less resources than replication-based approaches. These properties have led to the wide adoption of erasure codes in storage and communication systems [6, 43, 66–68, 72, 73, 91]. An erasure code encodes k data units to produce r redundant "parity" units in such a way that any k of the total $(k + r)$ data and parity units are sufficient for a decoder to recover the original k data units. The overhead incurred by an erasure code is $\frac{k+r}{k}$, which is typically much less than that of replication (by setting $r < k$).

A number of recent works have studied the theoretical aspects of using erasure codes for alleviating the effects of slowdowns and failures that occur in distributed *computation* (e.g.,[30, 54, 92]). This setup, called "coded-computation," uses erasure coding to recover the *outputs* of a deployed computation over data units. In coded-computation, data units are encoded into parity units, and the deployed computation is performed over all data and parity units in parallel. A decoder then uses the outputs from the fastest k of these computations to reconstruct the outputs corresponding to the original data units. For a prediction serving system, employing coded-computation would involve encoding *queries* such that a decoder can recover slow or failed *predictions*.

The primary differences between coded-computation and the traditional use of erasure codes in storage and communication come from (1) performing computation over encoded data and (2) the need for an erasure code to recover the results of computation over data units rather than the data units themselves. Whereas traditional applications of erasure codes involve encoding data units and decoding from a subset of data and parity units, in coded-computation one decodes by using the *output of computation over data and parity units*. This difference calls for fundamentally rethinking the design of erasure codes, as many of the erasure codes which have been widely used in storage and communication (e.g., Reed-Solomon codes [70]) are applicable only to a highly restricted class of computations [54].

As erasure codes can correct slowdowns with low latency and require less resource-overhead than replication-based techniques, enabling the use of coded-computation in prediction serving systems has potential for efficient mitigation of tail latency inflation. However, the complex non-linear components common to popular machine learning models, like neural networks, make it challenging to design effective coded-computation solutions for prediction serving systems. Existing coded-computation approaches, which focus on hand-crafting new erasure codes, can support only rudimentary computations [29, 30, 54, 57, 62, 71, 82, 92, 93], rendering them inadequate for prediction serving systems.

We propose to overcome the challenges of employing coded-computation for prediction serving systems via a *learning-based approach*. We show that machine learning can eschew the difficulty of hand-crafting codes and enable coded-computation over neural network inference. However, this approach requires careful consideration: we show that simply replacing encoders and decoders with machine learning models limits opportunities to reduce tail latency.

Motivated by these insights, we present *parity models*, a fundamentally new approach to coded-computation. A parity model is a neural network trained to convert encoded queries into a form that enables decoding of unavailable predictions. Unlike conventional coded-computation approaches, which design new erasure codes, parity models enable the use of simple, fast encoders and decoders, such as addition and subtraction. This reduces the computational burden introduced on a prediction serving system frontend, and also reduces the latency of reconstructions. We implement parity models in ParM (parity models), a prediction serving system designed to make use of erasure-coded resilience. As depicted in Figure 1, ParM encodes multiple queries together into a parity query. A parity model transforms the parity query such that its output enables a decoder to reconstruct slow or failed predictions.

The predictions returned by ParM are the same as any prediction serving system in the absence of slowdowns and failures. When slowdowns and failures do occur, the output of ParM's decoder is an approximate reconstruction of a slow or failed predictions. Reconstructing approximations of unavailable predictions is appropriate for inference, as predictions are already approximations and because ParM's reconstructions are returned only when a prediction from the deployed model would otherwise be slow or failed. In this scenario, it is preferable to return an approximate prediction rather than a late one or no prediction at all [15].

We have built ParM atop Clipper [25], an open-source prediction serving system. We evaluate the accuracy of ParM's

reconstructions on a variety of neural networks and inference tasks such as image classification, speech recognition, and object localization. We also evaluate ParM's ability to reduce tail latency across varying query rates, levels of background load, and amounts of redundancy. ParM reconstructs unavailable predictions with high accuracy and reduces tail latency while using 2-4× less additional resources than replication-based approaches. For example, using only half of the additional resources as replication, ParM's reconstructions from ResNet-18 models on various tasks are within a 6.5% difference in accuracy compared to if the original predictions were not slow or failed. This enables ParM to reign in tail latency: ParM reduces 99.9th percentile latency in the presence of load imbalance for a ResNet-18 model by up to 48% compared to a baseline that uses the same amount of resources as ParM, while maintaining the same median. This brings tail latency up to 3.5× closer to median latency, enabling ParM to maintain predictable latencies in the face of slowdowns and failures. These results show the promise of learning-based coded-computation to open new doors for imparting efficient resilience to prediction serving systems.

The code used to train and evaluate parity models is available at https://github.com/thesys-lab/parity-models.

2 Background and Motivation

This section describes prediction serving systems, as well as challenges and opportunities for improvement.

2.1 Prediction serving systems

A prediction serving system hosts machine learning models for inference; it accepts queries from clients, performs inference on hosted models, and returns predictions. We refer to a model hosted for inference as a "deployed model."

As depicted in Figure 1 (ignoring the purple components), prediction serving systems have two types of components: a frontend and model instances. The frontend receives queries and dispatches them to model instances for inference. Model instances are containers or processes that contain a copy of the deployed model and return predictions by performing inference on the deployed model.

Prediction serving systems employ scale-out architectures to serve predictions with low latency and high throughput and to overcome the memory and processing limitations of a single server [56]. In such a setup, multiple model instances are deployed on separate servers, each containing a copy of the same deployed model [25]. The frontend distributes queries to model instances according to a load-balancing strategy (e.g., single-queue, round-robin).

Prediction serving systems use a variety of hardware for inference, including GPUs [25], CPUs [39, 65, 96], TPUs [48], and FPGAs [23]. As some hardware is optimized for batched operation (e.g., GPUs), some systems will buffer queries at the frontend and dispatch them to model instances in batches [25, 64]. However, as buffering induces latency, many systems perform minimal or no batching [23, 96].

2.2 Challenges and opportunity

As described above, prediction serving systems are often run in a distributed fashion and make use of many cluster resources (e.g., compute, network). These systems are thus prone to the slowdowns and failures common to cloud and cluster settings. Left unmitigated, these slowdowns inflate tail latency and cause violations of latency targets. Prediction serving systems must therefore employ some means to mitigate the effects of slowdowns. Due to the many causes of slowdowns, such as those described in §1, it is important for mitigations to be agnostic to the cause of slowdowns.

A popular approach to imparting resilience to serving systems (e.g., web services) while remaining agnostic to the cause of slowdown is to issue redundant requests to multiple servers [26]. Previous work has theoretically motivated the potential for redundancy-based techniques to reduce latency [31, 47, 59, 76]. Redundancy-based approaches commonly navigate a tradeoff space between resource-overhead and latency. We next characterize existing redundancy-based techniques and where they operate within this tradeoff space.

Proactive approaches. A common technique used to achieve the lowest latency in recovering from slowdowns and failures is to proactively issue redundant requests to multiple servers and to wait only for the first replica to respond. Under such techniques, a system that replicates each query to d servers can tolerate $(d-1)$ slow/failed servers. By issuing redundant queries as soon as a query arrives in the system, proactive approaches mitigate slowdowns and failures with low latency. However, current proactive approaches result in high resource-overhead, as replicating queries to d servers requires d-times as many resources to handle increased load.

Reactive approaches. Reactive approaches operate with lower resource-overhead than replication-based approaches by issuing redundant queries only when confident that a slowdown or failure has occurred [13, 26, 94]. Under such approaches, each query is dispatched to a single server, and redundant requests are only issued if a certain amount of time has elapsed without receiving the result from the server. Such waiting time is commonly chosen to be long enough such that one is confident a server is running slowly if a result has not been returned by this time [26]. By issuing redundant queries only when confident that a slowdown or failure has occurred, reactive approaches reduce the amount of additional load introduced to a system, and thus the amount of resource-overhead necessary. However, by waiting for a significant amount of time before issuing redundant requests, reactive approaches are unable to mitigate slowdowns and failures with latency as low as proactive approaches.

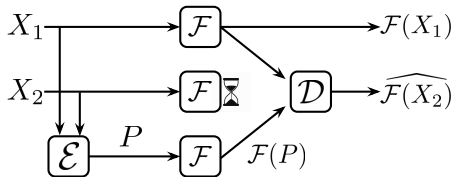


Figure 2. Abstract example of coded-computation with $k = 2$ original units and $r = 1$ parity units.

$\mathcal{F}(\mathbf{X})$	$\mathcal{F}(\mathbf{P})$	**Desired $\mathcal{F}(\mathbf{P})$**
$2X$	$2X_1 + 2X_2$	$2X_1 + 2X_2$
X^2	$X_1^2 + 2X_1X_2 + X_2^2$	$X_1^2 + X_2^2$

Table 1. Toy example with parity $P = X_1 + X_2$ showing the challenges of coded-computation on non-linear functions.

Erasure codes: proactive, resource-efficient. As described above, current proactive approaches operate with low-latency, but high resource-overhead, while current reactive approaches operate with low resource-overhead, but higher latency. Thus, existing techniques do not enable points of operation for serving systems other than the extremes in the tradeoff space between latency and resource-overhead.

Erasure codes are tools from the domain of coding theory capable of imparting proactive resilience to unavailability while using significantly less resource-overhead than replication-based approaches. These properties have made erasure codes an attractive and widely deployed alternative to replication-based and reactive approaches in storage and communication systems [6, 66, 68, 73, 91]. An erasure code encodes k data units to generate r redundant "parity units" such that any k out of the total $(k+r)$ data and parity units are sufficient for a decoder to recover the original k data units. Therefore, erasure codes operate with a resource-overhead of $\frac{k+r}{k}$, which is less than that of replication by setting $r < k$.

By varying parameters k and r, erasure codes enable fine-grained exploration of the tradeoff space between latency and resource-overhead in storage and communication systems. Thus, if possible, erasure codes have the potential to enable new points of operation within this tradeoff space that are currently inaccessible through existing proactive and reactive approaches for prediction serving systems.

2.3 Coded-computation and its challenges

The approach of using erasure codes for alleviating the effects of slowdowns and failures in *computation* is termed "coded-computation." Coded-computation differs fundamentally from the traditional use of erasure codes. Erasure codes have traditionally been used for recovering unavailable data units using a subset of data and parity units. In contrast, under coded-computation, (1) computation is performed over encoded data and (2) the goal is to recover unavailable *outputs of computation over data units* using a subset of the outputs of computation over data and parity units.

Example. Consider an example in Figure 2. Let $\mathcal{F}$ be a computation that is deployed on two servers. Let X_1 and X_2 be inputs to the computation. The goal is to return $\mathcal{F}(X_1)$ and $\mathcal{F}(X_2)$. In a prediction serving system, $\mathcal{F}$ is a deployed model and X_1 and X_2 are queries. Coded-computation adds an encoder $\mathcal{E}$ and a decoder $\mathcal{D}$, along with a third copy of $\mathcal{F}$ for tolerating one of the copies of $\mathcal{F}$ being unavailable. The encoder produces a parity $P = \mathcal{E}(X_1, X_2)$. The parity is dispatched to the third copy of $\mathcal{F}$ to produce $\mathcal{F}(P)$. Given $\mathcal{F}(P)$ and any one of $\{\mathcal{F}(X_1), \mathcal{F}(X_2)\}$, the decoder reconstructs the unavailable output. In the example in Figure 2, the second computation is slow. The decoder produces a reconstruction of $\mathcal{F}(X_2)$ as $\widehat{\mathcal{F}(X_2)} = \mathcal{D}(\mathcal{F}(X_1), \mathcal{F}(P))$.

General parameters. More generally, given k instances of $\mathcal{F}$, for k queries $X_1, \ldots, X_k$, the goal is to output $\mathcal{F}(X_1), \ldots, \mathcal{F}(X_k)$. To tolerate any r of these being unavailable, the encoder generates r parity queries that are operated on by r redundant copies of $\mathcal{F}$. The decoder acts on any k outputs of these $(k + r)$ instances of $\mathcal{F}$ to recover $\mathcal{F}(X_1), \ldots, \mathcal{F}(X_k)$.

Challenges. Coded-computation is straightforward when the underlying computation $\mathcal{F}$ is a *linear* function. A function $\mathcal{F}$ is linear if, for any inputs X_1 and X_2, and any constant a: (1) $\mathcal{F}(X_1 + X_2) = \mathcal{F}(X_1) + \mathcal{F}(X_2)$ and (2) $\mathcal{F}(aX_1) = a\mathcal{F}(X_1)$. Many of the erasure codes used in traditional applications, such as Reed-Solomon codes, can recover from unavailability of any linear function [54]. For example, consider having $k = 2$, $r = 1$. Suppose $\mathcal{F}$ is a linear function as in the first row of Table 1. Here, even a simple parity $P = X_1 + X_2$ suffices since $\mathcal{F}(P) = \mathcal{F}(X_1) + \mathcal{F}(X_2)$, and the decoder can subtract the available output from the parity output to recover the unavailable output. The same argument holds for any linear $\mathcal{F}$. However, a non-linear $\mathcal{F}$ significantly complicates the scenario. For example, consider $\mathcal{F}$ to be the simple non-linear function in the second row of Table 1. As shown in the table, $\mathcal{F}(P) \neq \mathcal{F}(X_1) + \mathcal{F}(X_2)$, and even for this simple function, $\mathcal{F}(P)$ involves complex non-linear interactions of the inputs which makes decoding difficult.

Coded-computation for inference? Handling non-linear computation is key to using coded-computation in prediction serving systems due to the non-linear components of popular machine learning models, like neural networks. While neural networks do contain linear components (e.g., matrix multiplication), they also contain many non-linear components (e.g., activation functions, max-pooling), which make the overall function computed by a neural network non-linear.

Existing techniques approach coded-computation by hand-crafting new encoders and decoders. However, due to the challenge of handling non-linear computations, these approaches support only rudimentary computations [29, 30,

54, 57, 62, 71, 82, 92, 93], and hence cannot support popular machine learning models like neural networks.

Thus, while coded-computation has promise for imparting resource-efficient resilience to slowdowns and failures, current approaches are inadequate for prediction-serving.

3 Handling Non-Linearity via Learning

We propose to overcome the barrier of coded-computation over non-linear functions via a *learning-based approach*. We next describe the potential and challenges of using machine learning to design a coded-computation framework.

3.1 Learning an erasure code

As illustrated in §2.3, it is challenging to hand-craft erasure codes for the non-linear components common to neural networks. This problem is further complicated by the multitude of mathematical components employed in neural networks (e.g., types of activation functions); even if one hand-crafted an erasure code suitable for one neural network, the approach might not work for other neural networks.

To overcome this, we make a key observation: erasure codes for coded-computation can be *learned*. Using machine learning models for encoders and decoders, designing an erasure code simply involves training encoder and decoder models. Consider again the example in Figure 2. An optimization problem for learning encoder $\mathcal{E}$ and decoder $\mathcal{D}$ for this example is: given $\mathcal{F}$, train $\mathcal{E}$ and $\mathcal{D}$ so as to minimize the difference between $\widehat{\mathcal{F}(X_2)}$ and $\mathcal{F}(X_2)$, for all pairs (X_1, X_2), with $\widehat{\mathcal{F}(X_2)} = \mathcal{D}(\mathcal{F}(X_1), \mathcal{F}(P))$ and $P = \mathcal{E}(X_1, X_2)$.

One distinction of using learned encoders and decoders as opposed to hand-crafted ones is that reconstructions of unavailable outputs will be *approximations* of the function outputs that would be returned if they were not slow or failed. However, any decrease in accuracy due to reconstruction is only incurred in the case when a model is slow to return a prediction. In this scenario, prediction services prefer to return an approximate prediction rather than a late one [15].

3.2 Benefits and challenges of learned codes

Based on this observation, we developed learned encoders and decoders for coded-computation over neural networks. We focused on designing encoders and decoders as neural networks due to their recent success in a number of tasks. Our experimentation with this approach was nuanced; as this approach is not the primary focus of this paper, we only describe the experimental outcomes. For further details on this approach, we direct the reader to our technical report [50].

We experimented with many neural networks for encoders and decoders on image classification tasks. We found that a particular encoder and decoder pair was capable of reconstructing unavailable predictions from a ResNet-18 model on the CIFAR-10 dataset with 82% accuracy. This is a small drop in accuracy compared to the accuracy of the deployed model (93%), considering that the drop only occurs when the deployed model is unavailable. This marks a significant step forward from existing coded-computation techniques, which are unable to support even simple neural networks.

While our experience with learned erasure codes showcased the potential of learning-based coded-computation, it also revealed a challenge: neural network encoders and decoders add significant latency to reconstruction. Recall that a coded-computation technique can reconstruct unavailable outputs only after (1) encoding, (2) performing k out of $(k+r)$ computations, and (3) decoding. As neural networks are often computationally expensive, encoding and decoding with neural networks adds significant latency to this process, and thus limits opportunities for tail latency reduction. Indeed, we found that the average latency of encoding and decoding using the most accurate models was up to 7× higher than that of the deployed model, making this approach appropriate only for alleviating extreme slowdowns. Furthermore, using computationally expensive neural network encoders and decoders would require hardware acceleration and a beefier, expensive frontend, which is the ideal location for encoders and decoders, as will be described in §4.1.

Takeaway. Our experience with learning erasure codes indicates that using machine learning has promise for overcoming the challenges of coded-computation over non-linear functions, but that learned erasure codes can significantly increase the latency of reconstructing unavailable predictions, limiting the effectiveness in reducing tail latency.

These insights motivate ParM's novel approach to coded-computation, which we describe next.

4 Design of ParM

To overcome the challenge of performing coded-computation over non-linear functions as well as the overhead of learned erasure codes, we take a fundamentally new approach to coded-computation. Rather than designing new encoders and decoders, we propose to use simple, fast encoders and decoders and instead design a *new computation over parities*, called a "parity model." We implement this approach in ParM, a prediction serving system that uses parity models to enable erasure-coded resilience. In the example in Figure 2, instead of the extra copy of $\mathcal{F}$ deployed by current coded-computation approaches, ParM introduces a parity model, which we denote as $\mathcal{F}_P$. The challenge of this approach is to design a parity model that enables reconstruction. We address this by designing parity models as neural networks, and learning a parity model that enables simple encoders and decoders to reconstruct slow or failed predictions.

By learning a parity model and using simple, fast encoders and decoders, ParM is (1) able to impart resilience to modern machine learning models, like neural networks, while (2) operating with low latency without requiring expensive hardware acceleration for encoding and decoding.

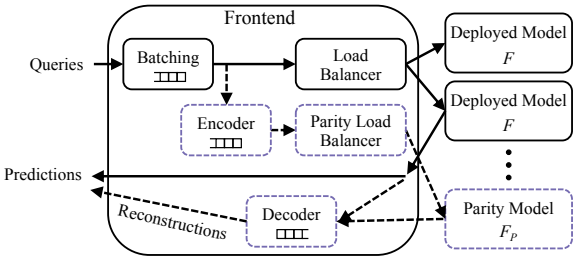


Figure 3. Components of a prediction serving system and those added by ParM (dotted). Queues indicate components which may group queries/predictions (e.g., coding group).

Setting and notation. We first describe ParM in detail for imparting resilience to any one out of k predictions experiencing slowdown or failure (i.e., $r = 1$). This setting is motivated by measurements of production clusters [67, 68]. Section 7 describes how the proposed approach can tolerate multiple unavailabilities (i.e., $r > 1$) as well. We will continue to use the notation of $\mathcal{F}$ to represent the deployed model, X_i to represent a query, $\mathcal{F}(X_i)$ to represent a prediction resulting from inference on $\mathcal{F}$ with X_i, and $\widehat{\mathcal{F}(X_i)}$ to represent a reconstruction of $\mathcal{F}(X_i)$ when $\mathcal{F}(X_i)$ is unavailable.

4.1 System architecture

The architecture of ParM is shown in Figure 3. ParM builds atop a typical prediction serving system architecture that has m instances of a deployed model. Queries sent to the frontend are batched (according to a batching policy) and dispatched to a model instance for inference on the deployed model. Query batches[1] are dispatched to model instances according to a provided load-balancing strategy.

ParM adds an encoder and a decoder on the frontend along with $\frac{m}{k}$ instances of a parity model. Each parity model uses the same amount of resources (e.g., compute, network) as a deployed model. ParM thus adds $\frac{1}{k}$ resource-overhead.

As query batches are dispatched, they are placed in a *coding group* consisting of k batches that have been consecutively dispatched. A coding group acts similarly to a "stripe" in erasure-coded storage systems; the query batches of a coding group are encoded to create a single "parity batch." ParM treats queries as though they are independent from one another. In particular, the queries that form a coding group need not have been sent by the same application or user. Encoding takes place across individual queries of a coding group: the ith queries of each of the k query batches in a coding group are encoded to produce the ith query of the parity batch. Encoding *does not* delay query dispatching as query batches are immediately handled by the load balancer when they are formed, and placed in a coding group for later encoding. The parity batch is dispatched to a parity model and the output resulting from inference over the parity model is returned to the frontend. Encoding is performed on the frontend rather than on a parity model so as to incur only $\frac{1}{k}$ network bandwidth overhead. Otherwise, all queries would need to be replicated to a parity model prior to encoding, which would incur 2× network bandwidth overhead.

Predictions that are returned to the frontend are immediately returned to clients. ParM's decoder is only used when any one of the k prediction batches from a coding group is unavailable. ParM enables flexibility in determining when a prediction is considered unavailable, and thus when decoding is necessary. A prediction could be considered unavailable if the $(k-1)$ other predictions from its coding group and the output of the parity model have been returned before the prediction in question. Alternatively, a system could set a timeout after which a prediction is considered unavailable if the conditions above are still not met.

When a prediction is considered unavailable, the decoder uses the outputs of the parity model and the $(k-1)$ available model instances to reconstruct an approximation of the unavailable prediction batch. Approximate predictions are returned only when predictions from the deployed model are unavailable. ParM thus reduces tail latency when an unavailable prediction is reconstructed before the actual prediction for the query returns (e.g., from a slow model instance).

4.2 Encoder and decoder

Introducing and learning parity models enables ParM to use simple, fast erasure codes to reconstruct unavailable predictions. ParM can support many different encoder and decoder designs, opening up a rich design space. In this paper, we will illustrate the power of parity models by using the simple addition/subtraction erasure code described in §2.3, and showing that even with this simple encoder and decoder, ParM significantly reduces tail latency and accurately reconstructs unavailable predictions. This simple encoder and decoder is applicable to a wide range of inference tasks, including image classification, speech recognition, and object localization, allowing us to showcase ParM's applicability to a variety of inference tasks. A prediction serving system that is specialized to a specific inference task could potentially benefit from designing task-specific encoders and decoders for use in ParM, such as an encoder that downsamples and concatenates image queries for image classification. We evaluate an example of such a task-specific code in §5.2.3.

Under the simple addition/subtraction encoder and decoder showcased in this paper, the encoder produces a parity as the summation of queries in an coding group, i.e., $P = \sum_{i=1}^{k} X_i$. Queries are normalized to a common size prior to encoding, and summation is performed across corresponding features of each query (e.g., top-right pixel of each image query). The decoder subtracts $(k-1)$ available predictions from the output of the parity model $\mathcal{F}_P(P)$ to reconstruct

[1] We use the terms "batch" and "query batch" to refer to one or more queries dispatched to a model instance at a single point in time.

an unavailable prediction. Thus, an unavailable prediction $\mathcal{F}(X_j)$ is reconstructed as $\widehat{\mathcal{F}(X_j)} = \mathcal{F}_P(P) - \sum_{i \neq j}^{k} \mathcal{F}(X_i)$.

4.3 Parity model design

ParM uses neural networks for parity models to learn a model that transforms parities into a form that enables decoding. In order for a parity model to help in mitigating slowdowns, the average runtime of a parity model should be similar to that of the deployed model. One simple way of enforcing this is by using the same neural network architecture for the parity model as is used for the deployed model (i.e., same number and size of layers). Thus, if the deployed model is a ResNet-18 architecture, the parity model also uses ResNet-18, but with parameters trained using the procedure that will be described in §4.4. As a neural network's architecture determines its runtime, this approach ensures that the parity model has the same average runtime as the deployed model. We use this approach in our evaluations.

In general, a parity model is not required to use the same architecture as the deployed model. In cases where it is necessary or preferable to use a different architecture for a parity model, such as when the deployed model is not a neural network, a parity model could be designed via architecture search [97]. However, we do not focus on this scenario.

It is common in classification tasks to use a softmax function to convert the output of a neural network into a probability distribution. We do not apply a softmax function to the output of a parity model, as the desired output of a parity model is not necessarily constrained to be a probability distribution. As we will describe in §4.4, we employ a loss function in training that does not require the output of parity model to be a probability distribution.

4.4 Training a parity model

A parity model is trained prior to being deployed.

Training data. The training data are the parities generated by the encoder, and training labels are the transformations expected by the decoder. For the simple encoder and decoder described in §4.2, with $k = 2$, training data from queries X_1 and X_2 are $(X_1 + X_2)$ and labels are $(\mathcal{F}(X_1) + \mathcal{F}(X_2))$.

Training data is generated using queries that are representative of those issued to the deployed model for inference. A parity model is trained using the same dataset used for training the deployed model, whenever available. Thus, if the deployed model was trained using the CIFAR-10 [16] dataset, samples from CIFAR-10 are used as queries $X_1, \ldots, X_k$ that are encoded together to generate training samples for the parity model. Labels are generated by performing inference with the deployed model to obtain $\mathcal{F}(X_1), \ldots, \mathcal{F}(X_k)$ and summing these predictions to form the desired parity model output. For example, if the outputs of a deployed model are vectors of n floating points, as is the case in a classification task with n classes, a label would be generated as the element-wise summation of these vectors. ParM can also use as labels the summation of the true labels for queries.

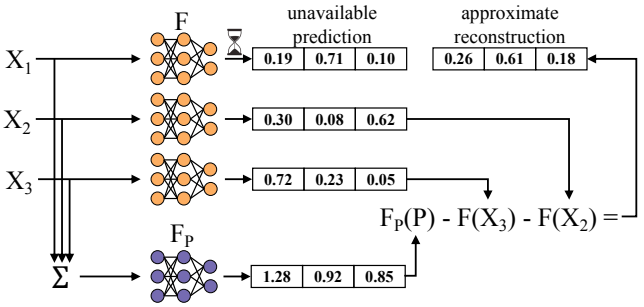


Figure 4. Example of ParM ($k = 3$) mitigating a slowdown.

If the dataset used for training the deployed model is not available, a parity model can be trained using queries that have been issued to ParM for inference on the deployed model. The predictions that result from inference on the deployed model are used to form labels for the parity model. In this case, as expected, ParM can deliver benefits only after the parity model has been trained to a sufficient degree.

Loss function. While there are many loss functions that could be used in training a parity model, we use the mean-squared-error (MSE) between the output of the parity model and the desired output as the loss function. We choose MSE rather than a task-specific loss function (e.g., cross-entropy) to make ParM applicable to many inference tasks.

Training procedure. Training a parity model involves the same optimization procedure commonly used for training neural networks. In each iteration, k samples are drawn at random from the deployed model's training dataset and encoded to form a parity sample. The parity model performs inference over this parity query (forward pass) to generate $\mathcal{F}_P(P)$. A loss value for this parity query is calculated between $\mathcal{F}_P(P)$ and the desired parity model output (e.g., $\mathcal{F}(X_1) + \mathcal{F}(X_2)$ for the addition/subtraction code with $k = 2$). Parity model parameters are updated based on this loss value using the standard backpropogation algorithm (backward pass). This iterative process continues until a parity model reaches sufficient accuracy on a validation dataset.

Reducing the time to train a parity model was not a goal of this work; we describe inefficiencies of the procedure above and their potential solutions in §7.

4.5 Example

Figure 4 shows an example of how ParM mitigates unavailability of any one of three model instances (i.e., $k = 3$). Queries X_1, X_2, X_3 are dispatched to three separate model instances for inference on deployed model $\mathcal{F}$ to return predictions $\mathcal{F}(X_1), \mathcal{F}(X_2), \mathcal{F}(X_3)$. The learning task here is classification across n classes. Each $\mathcal{F}(X_i)$ is thus a vector of

n floating-points ($n = 3$ in Figure 4). As queries are dispatched to model instances, they are encoded (Σ) to generate a parity $P = (X_1 + X_2 + X_3)$. The parity is dispatched to a parity model $\mathcal{F}_P$ to produce $\mathcal{F}_P(P)$. In this example, the model instance processing X_1 is slow. The decoder reconstructs this unavailable prediction as $(\mathcal{F}_P(P) - \mathcal{F}(X_3) - \mathcal{F}(X_2))$. The reconstruction provides a reasonable approximation of the true prediction that would have been returned had the model instance not been slow (labeled as "unavailable prediction").

In this example, ParM remains resilient to any one out of three model instances being unavailable by using one extra instance to serve a parity model. To achieve the same resilience, a replication-based approach requires three extra model instances. Thus, in this example, ParM operates with 3× less additional resources than replication-based approaches. More generally, ParM operates with k-times less additional resources of replication-based approaches.

5 Evaluation of Accuracy

In this section, we evaluate ParM's ability to accurately reconstruct unavailable predictions.

5.1 Experimental setup

We use PyTorch [12] to train separate parity models for each parameter k, dataset, and deployed model.

Inference tasks and models. Learning a parity model to enable reconstruction of predictions represents a fundamentally new learning task. Therefore, we evaluate ParM on popular inference tasks and datasets to establish the potential of using parity models. Specifically, we use popular image classification (CIFAR-10 and 100 [16], Cat v. Dog [14], Fashion-MNIST [87], and MNIST [52]), speech recognition (Google Commands [85]), and object localization (CUB-200 [86]) tasks. We evaluate ParM on the ImageNet dataset [74] in §5.2.3.

As described in §4.3, a parity model uses the same neural network architecture as the deployed model. We consider five different architectures: a multi-layer perception (MLP),[2] LeNet-5 [53], VGG-11 [77], ResNet-18, and ResNet-152 [40]. The former two are simpler neural networks while the others are variants of state-of-the-art neural networks.

Parameters. We consider values for parameter k of 2, 3, and 4, corresponding to 33%, 25%, and 20% redundancy. We use Adam [27], learning rate of 0.001, L2-regularization of 10^{-5}, and batch sizes between 32-64. Convolutional layers use Xavier initialization [32], biases are initialized to zero, and other weights are initialized from $\mathcal{N}(0, 0.01)$.

Encoder and decoder. Unless otherwise specified, we use the generic addition encoder and subtraction decoder described in §4.2. We showcase the benefit of employing task-specific encoders and decoders within ParM in §5.2.3.

[2]The MLP has two hidden layers with 200 and 100 units and ReLUs.

Metrics. Analyzing erasure codes for storage and communication involves reasoning about performance under normal operation (when unavailability does not occur) and performance in "degraded mode" (when unavailability occurs and reconstruction is required). These different modes of operation are similarly present for inference. The overall accuracy of any approach is calculated based on its accuracy when predictions from the deployed model are available (A_a) and its accuracy when these predictions are unavailable (A_d, "degraded mode"). If f fraction of deployed model predictions are unavailable, the overall accuracy (A_o) is:

$$A_o = (1 - f)A_a + fA_d \quad (1)$$

ParM aims to achieve high A_d; it does not change the accuracy when predictions from the deployed model are available (A_a). We report both A_o and A_d.

We report accuracies on test datasets, which are not used in training. Test samples are randomly placed into groups of k and encoded to produce a parity. For each parity P, we compute the output of inference on the parity model as $\mathcal{F}_P(P)$. During decoding, we use $\mathcal{F}_P(P)$ to reconstruct $\widehat{\mathcal{F}(X_i)}$ for each X_i that was used in encoding P, simulating every scenario of one prediction being unavailable. Each $\widehat{\mathcal{F}(X_i)}$ is compared to the true label for X_i. For CIFAR-100, we report top-5 accuracy, as is common (i.e., the fraction for which the true class of X_i is in the top 5 of $\widehat{\mathcal{F}(X_i)}$).

5.2 Results

Figure 5 shows the accuracy of the deployed model (A_a) along with the degraded mode accuracy (A_d) of ParM with $k = 2$ for image classification and speech recognition tasks using ResNet-18. VGG-11 is used for the speech dataset and ResNet-152 for CIFAR-100. ParM's degraded mode accuracy is no more than 6.5% lower than that when predictions from the deployed model are available. As Figure 6 illustrates, this enables ParM to maintain high overall accuracy (A_o) in the face of unavailability. For this example, at expected levels of unavailability (i.e., f less than 10%), ParM's overall accuracy is at most 0.4%, 1.9%, and 4.1% lower than when all predictions are available at k values of 2, 3, and 4, respectively. This indicates a tradeoff between ParM's parameter k, which controls resource-efficiency and resilience, and the accuracy of reconstructed predictions, which we discuss in §5.2.2. Finally, ParM's framework enables encoders and decoders to be specialized to the inference task at hand, allowing for further increase in degraded mode accuracy. We showcase task-specific specialization with an image-classification-specific encoder on CIFAR and ImageNet datasets (§5.2.3).

5.2.1 Inference tasks

ParM achieves high degraded mode accuracy with $k = 2$ for the image classification and speech recognition datasets in Figure 5. For these tasks, degraded mode accuracy is at

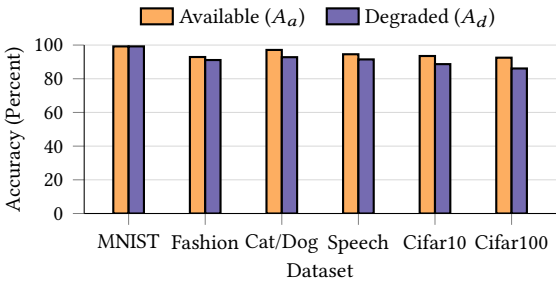


Figure 5. Comparison of ParM's accuracy when predictions from the deployed model are available (A_a, "Available") and when ParM's reconstructions are necessary (A_d, "Degraded"). ParM uses $k = 2$.

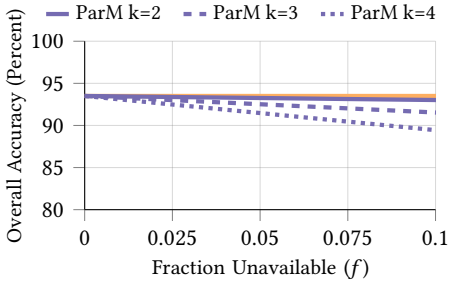


Figure 6. Overall accuracy (A_o) of predictions on CIFAR-10 as the fraction of predictions that are unavailable (f) increases. The horizontal orange line shows the accuracy of the ResNet-18 deployed model (A_a).

Figure 7. Example of ParM's reconstruction for object localization.

most 6.5% lower than when predictions are not slow or failed. We observe similar results for a variety of neural network architectures. For example, on the Fashion-MNIST dataset, ParM's degraded mode accuracy for the MLP, LeNet-5, and ResNet-18 models are only 1.7-9.8% lower than the accuracy when predictions from the deployed model are available.

Object localization task. We next evaluate ParM on object localization, which is a regression task. The goal in this task is to predict the coordinates of a bounding box surrounding an object of interest in an image. As a proof of concept of ParM's applicability for this task, we evaluate ParM on the Caltech-UCSD Birds dataset [86] using ResNet-18. The performance metric for localization tasks is the intersection over union (IoU): the IoU between two bounding boxes is computed as the area of their intersection divided by the area of their union. IoU values fall between 0 and 1, with an IoU of 1 corresponding to identical boxes, and an IoU of 0 corresponding to boxes with no overlap.

Figure 7 shows an example of the bounding boxes returned by the deployed model and ParM's reconstruction. For this example, the deployed model has an IoU of 0.88 and ParM's reconstruction has an IoU of 0.61. ParM's reconstruction captures the gist of the localization and would serve as a reasonable approximation in the face of unavailability. On the entire dataset, the deployed model achieves an average IoU of 0.95 with ground-truth bounding boxes. In degraded mode, ParM with $k = 2$ achieves an average IoU of 0.67.

5.2.2 Varying redundancy via parameter k

Figure 8 shows ParM's degraded mode accuracy with $k = 2, 3, 4$. ParM's degraded mode accuracy decreases with increasing parameter k. As parameter k increases, features from more queries are packed into a single parity query, making the parity query noisier and making it challenging to learn a parity model. This indicates a tradeoff between the value of parameter k and degraded mode accuracy.

To put these accuracies in perspective, consider a scenario in which no redundant computation is used to mitigate unavailability. In this case, when the output from any deployed model is unavailable, the best one can do is to return a random output as a "default" prediction. The option to provide such a default prediction is available in Clipper. The degraded mode accuracy when returning default predictions depends on the number of possible outputs of an inference task (e.g., the number of classes). For example, on a classification task with ten classes, the expected degraded mode accuracy of this technique would be 10%. The degraded mode accuracy of default predictions provides a lower bound on degraded mode accuracy and an indicator of the difficulty of a particular inference task. For all datasets in Figure 8, ParM's degraded mode accuracy is significantly above this lower bound, indicating that ParM makes significant progress in the task of reconstructing predictions.

5.2.3 Inference task-specific encoders and decoders

As described in §4.2, ParM's framework enables a large design space for possible encoders and decoders. So far, all evaluation results have used the simple, general addition encoder and subtraction decoder, which is applicable to many inference tasks. We now showcase the breadth of ParM's framework by evaluating ParM's accuracy with alternate encoders and decoders that are inference-task specific.

We design an encoder specialized for image classification which takes in k image queries, and downsizes and concatenates them into a parity query. For example, as shown in Figure 9, given $k = 4$ images from the CIFAR-10 dataset (each with $32 \times 32 \times 3$ features), each image is resized to have $16 \times 16 \times 3$ features and concatenated together. The resulting parity query is a 2×2 grid of these resized images, and thus has a total of $32 \times 32 \times 3$ features, the same amount as a single image query. We use the subtraction decoder alongside this encoder. In training parity models with this encoder, we use a loss function specialized to classification: the cross-entropy

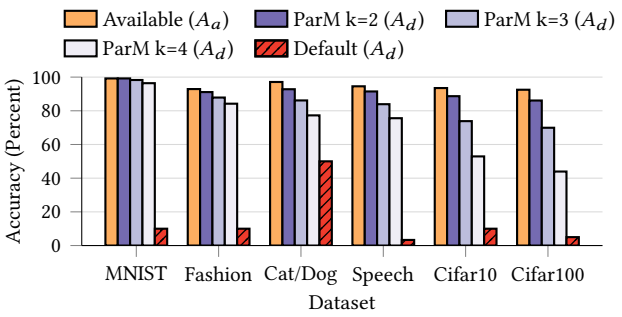


Figure 8. Accuracies of predictions reconstructed by ParM with $k = 2, 3, 4$ compared to returning a default response when deployed model predictions are unavailable (A_d).

between the output of the parity model and the summation of the one-hot-encoded labels for concatenated images (after normalizing this summation to be a probability distribution).

By specializing to the task at hand, this encoder improves degraded mode accuracy compared to the general addition encoder. For example, at k values of 2 and 4 on CIFAR-10, the task-specific encoder achieves a degraded mode accuracy of 89% and 74%, respectively. This represents a 2% and 22% improvement compared to the general encoder, respectively.

On the 1000-class ImageNet dataset (ILSVRC 2012 [74]) with $k = 2$ and using ResNet-50 models, this approach achieves a 61% top-5 degraded mode accuracy. To put these results in perspective, we note that the first use of neural networks for the ImageNet classification task resulted in a top-5 accuracy of 84.7% [51]. Our results similarly represent the first use of learning and neural networks for coded-computation. As the task of a parity model is considerably more difficult than that of image classification, these results show the promise of using parity models for coded computation, even on massive datasets. We expect that further improvements may be achieved on large datasets like ImageNet through a more principled approach to forming training samples than the random selection described in §4.4. This will be particularly beneficial for classification datasets like ImageNet for which there is imbalance in the number of training samples available per class. We also expect that improvement in degraded mode accuracy may be achieved by further exploring the design space for encoders, decoders, and the model architecture used for parity models.

6 Evaluation of Tail Latency Reduction

We next evaluate ParM's ability to reduce tail latency. The highlights of the evaluation results are as follows:

- ParM significantly reducing tail latency: in the presence of load imbalance, ParM reduces 99.9th percentile latency by up to 48%, bringing tail latency up to 3.5× closer to median latency, while maintaining the same median (§6.2.1). Even with little load imbalance, ParM reduces the gap between tail and median latency by up to 2.3× (§6.2.4). These benefits hold for a variety of inference hardware, query rates, and batch sizes.
- ParM's approach of introducing and learning parity models enables using encoders and decoders with low latencies (less than 200 µs and 20 µs, respectively) (§6.2.5).
- ParM reduces tail latency while maintaining simpler development and deployment than other hand-crafted approaches, such as deploying approximate models (§6.2.6).

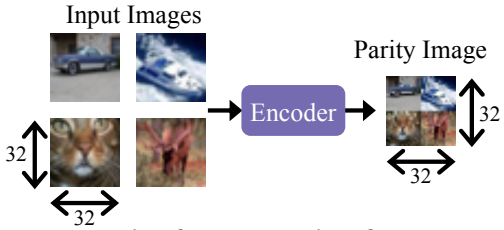


Figure 9. Example of an image-classification-specific encoder with $k = 4$ on the CIFAR-10 dataset.

6.1 Implementation and Evaluation Setup

We have built ParM atop Clipper [25], an open-source prediction serving system. We implement ParM's encoder and decoder on the Clipper frontend in C++. Inference runs in Docker containers on model instances, as is standard in Clipper, and we use PyTorch [12] to implement models. We disable the prediction caching feature in Clipper to evaluate end-to-end latency, though ParM does not preclude the use of prediction caching. We use OpenCV [11] for pixel-level encoder operations. We use the addition encoder and subtraction decoder described in §5.2.3 in all latency evaluations.

Baselines. We consider as a baseline a prediction serving system with the same number of instances as ParM but using all additional instances for deploying extra copies of the deployed model. We call this baseline "Equal-Resources." For a setting of parameter k on a cluster with m model instances for deployed models, both ParM and Equal-Resources use $\frac{m}{k}$ additional model instances. ParM uses these extra instances to deploy parity models, whereas this baseline hosts extra deployed models on these instances. These extra instances enable the baseline to reduce system load, which reduces tail latency and provides a fair comparison. We compare ParM to another baseline, deploying approximate models, in §6.2.6.

Cluster setup. All experiments are run on Amazon EC2. We evaluate ParM on two different cluster setups to mimic various production prediction-serving settings.

- **GPU cluster.** Each model instance is a `p2.xlarge` instance with one NVIDIA K80 GPU. We use 12 instances for deployed models and $\frac{12}{k}$ additional instances for redundancy. With $k = 2$ there are thus 18 instances.
- **CPU cluster.** Each model instance is a `c5.xlarge` instance, which AWS recommends for inference [2]. We use 24 instances for deployed models and $\frac{24}{k}$ additional

instances for redundancy. This emulates production settings that use CPUs for inference [39, 65, 96]. This cluster is larger than the GPU cluster since the CPU instances are less expensive than GPU instances.

We use a single frontend of type `c5.9xlarge`. We use this larger instance for the frontend to sustain high aggregate network bandwidth to model instances (10 Gbps). Each instance uses AWS ENA networking. We observe bandwidth of 1-2 Gbps between each GPU instance and the frontend and of 4-5 Gbps between each CPU instance and the frontend.

Queries and deployed models. Recall that accuracy results were presented for various tasks and deployed models in §5. For latency evaluations we choose one of these models and tasks, ResNet-18 [40] for image classification. We use ResNet-18 rather than a larger model like ResNet-152, which would have a longer runtime, to provide a more challenging scenario in which ParM must reconstruct predictions with low latency. Queries are drawn from the Cat v. Dog [14] dataset. These higher-resolution images test the ability of ParM's encoder to operate with low latency.[3] We modify deployed models and parity models to return 1000 values as predictions to create a more computationally challenging decoding scenario in which there are 1000 classes in each prediction, rather than the usual 2 classes for this task.

Load balancing. Both ParM and the baseline use a single-queue load balancing strategy for dispatching queries to model instances as is used in Clipper, and is optimal in reducing average response time [36]. The frontend maintains a single queue to which all queries are added. Model instances pull queries from this queue when they are available. Similarly, ParM adds parity queries to a single queue which parity models pull from. Evaluation on other, sub-optimal, load balancing strategies (e.g., round-robin) revealed results that are even more favorable for ParM than those showcased below.

Background traffic. Like any redundancy-based technique, ParM is most beneficial in operating scenarios prone to unpredictable latency spikes and failures; if unavailability is absent or entirely predictable, redundancy-based approaches are not necessary. Therefore, we focus the evaluation of ParM on these scenarios by inducing background load on the clusters running ParM. The main form of background load we use emulates network traffic typical of data analytics workloads. Specifically, two model instances are chosen randomly to transfer data to one another of size randomly drawn between 128-256 MB. Unless otherwise mentioned, four shuffles take place concurrently. In this setting *only* the cluster network is imbalanced; we do not introduce computational multitenancy. We experiment with light multitenant computation and varying the number of shuffles in §6.2.4.

[3]While CIFAR-10/100 are more difficult tasks for training a model than Cat v. Dog, their low resolution makes them computationally inexpensive. This makes Cat v. Dog a more realistic workload for evaluating latency.

Latency metric. Clients send 100-thousand queries to the frontend using a variety of Poisson arrival rates. All latencies measure the time between when the frontend receives a query and when the corresponding prediction is returned to the frontend (from a deployed model or reconstructed). We report the median of three runs of each configuration, with error bars showing the minimum and maximum. Unless otherwise noted, all experiments are run with batch size of one, as this is the preferred batch size for low latency [23, 96]. We evaluate ParM with larger batch sizes in §6.2.3.

6.2 Results

We now report ParM's reduction of tail latency.

6.2.1 Varying query rate

Figure 10 shows median and 99.9th percentile latencies with $k = 2$ (i.e., both ParM and Equal-Resources have 33% redundancy) on the GPU and CPU clusters. We consider query rates up until a point in which a prediction serving system with no redundancy (i.e., with m instances) experiences tail latency dominated by queueing. Beyond this point, ParM could be used alongside techniques that reduce queueing [24].

ParM reduces the gap between 99.9th percentile and median latency by 2.6-3.2× compared to Equal-Resources on the GPU cluster, and by 3-3.5× on the CPU cluster. ParM's 99.9th percentile latencies are thus 38-43% lower on the GPU cluster and 44-48% lower on the CPU cluster. This enables ParM to operate with more predictable latency. As expected from any redundancy-based approach, ParM adds additional system load by issuing redundant queries, leading to a slight increase in median latency (less than 0.5 ms, which is negligible).

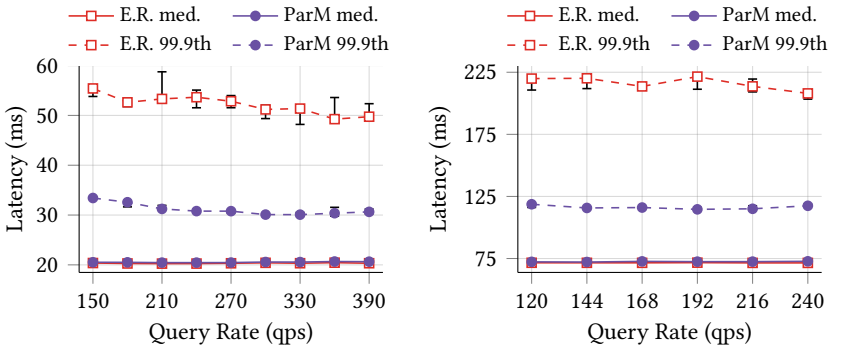
6.2.2 Varying redundancy via parameter k

Figure 11 shows the latencies achieved by ParM with k being 2, 3, and 4, when operating at 270 qps on the GPU cluster. As k increases, ParM's tail latency also increases. This is due to two factors. First, at higher values of k, ParM is more vulnerable to multiple predictions in a coding group being unavailable, as the decoder requires $k-1$ predictions from the deployed model to be available (in addition to the output of the parity model). Second, increasing k increases the amount of time ParM needs to wait for k queries to arrive before encoding into a parity query. This increases the latency of the end-to-end path of reconstructing an unavailable prediction.

Despite these factors, ParM still reduces tail latency, even when using less resources than the baseline. At k values of 3 and 4, which have 25% and 20% redundancy, ParM reduces the gap between tail and median latency by up to 2.5× compared to when Equal-Resources has 33% redundancy.

6.2.3 Varying batch size

Due to the low latencies required by user-facing applications, many prediction serving systems perform no or minimal query batching [23, 39, 96]. For completeness, we evaluate



(a) GPU cluster **(b)** CPU cluster

Figure 10. Latencies of ParM and Equal-Resources (E.R.). The CPU cluster has twice as many instances as the GPU cluster and thus sustains comparable load.

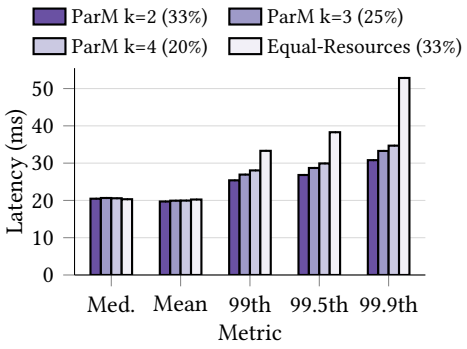


Figure 11. Latencies of ParM at varying values of k compared to the strongest baseline. The amount of redundancy used in each configuration is listed in parentheses.

ParM when queries are batched on the GPU cluster. ParM uses $k = 2$ in these experiments and query rate is set to 460 qps and 584 qps for batch sizes of 2 and 4, respectively. These query rates are obtained by scaling from 300 qps used at batch size 1 based on the throughput improvement observed with increasing batch sizes. ParM outperforms Equal-Resources at all batch sizes: at batch sizes of 2 and 4, ParM reduces 99.9th percentile latency by 43% and 47%, respectively.

6.2.4 Varying degrees and types of load imbalance

All experiments so far were run with background network imbalance, as described in §6.1. ParM reduces tail latency even with lighter background network load: Figure 12 shows that when 2 and 3 concurrent background shuffles take place (as opposed to the 4 used for most experiments), ParM reduces 99.9th percentile latency over Equal-Resources by 35% and 39%, respectively on the GPU cluster with query rate of 270 qps. ParM's benefits increase with higher load imbalance, as ParM reduces the gap between 99.9th and median latency by 3.5× over Equal-Resources with 5 background shuffles.

To evaluate ParM's resilience to a different, lighter form of load imbalance, we run light background inference tasks on model instances. Specifically, we deploy ResNet-18 models on one ninth of instances using a separate copy of Clipper, and send an average query rate of less than 5% of what the cluster can maintain. We do not add network imbalance in this setting. Figure 13 shows latencies at $k = 2$ on the GPU cluster with varying query rate. Even with this light form of imbalance, ParM reduces the gap between 99.9th percentile and median latency by up to 2.3× over Equal-Resources.

6.2.5 Latency of ParM's components

ParM's latency of reconstructing unavailable predictions consists of encoding, parity model inference, and decoding. ParM has median encoding latencies of 93 µs, 153 µs, and 193 µs, and median decoding latencies of 8 µs, 14 µs, and 19 µs for k values of 2, 3, and 4, respectively. As the latency of parity model inference is tens of milliseconds, ParM's encoding and decoding make up a very small fraction of end-to-end reconstruction latency. These fast encoders and decoders are enabled by introducing and learning parity models.

6.2.6 Comparison to approximate backup models

An alternative to ParM is to replace parity models with less computationally expensive models that approximate the predictions of the deployed model, and to replicate queries to these approximate models. This approach has a number of drawbacks: (1) it is unstable at expected query rates, (2) it is inflexible to changes in hardware, limiting deployment flexibility, and (3) it requires 2× network bandwidth. To showcase these drawbacks of the alternative approach, we compare ParM (with $k = 2$) to the aforementioned alternative using $\frac{m}{k}$ extra model instances for approximate models. We use MobileNet-V2 [75] (with a width factor of 0.25) as the approximate models because this model has similar accuracy (87.6%) as ParM's reconstructions (87.4%) for CIFAR-10.

Figure 14 shows the latencies of these approaches on the GPU cluster with varying query rate. While ParM's 99.9th percentile latency varies only modestly, using approximate models results in tail latency variations of over 36%. This variance occurs because all queries are replicated to approximate models even though there are only $\frac{1}{k}$ as many approximate models as there are deployed models. Thus, approximate models must be k-times faster than the deployed model for this system to be stable. The approximate model in this case is not k-times faster than the deployed model, leading to inflated tail latency due to queueing as query rate increases.

Even if one crafted an approximate model satisfying the runtime requirement described above, the model may not be appropriate for different hardware. We find that the speedup achieved by the approximate model over the deployed model varies substantially across different inference hardware. For example, the MobileNet-V2 approximate model is 1.4× faster than the ResNet-18 deployed model on the CPU cluster, but

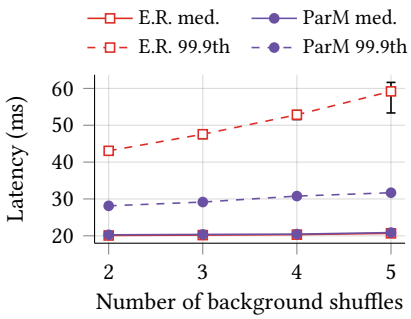


Figure 12. ParM and Equal-Resources (E.R.) with varying network imbalance.

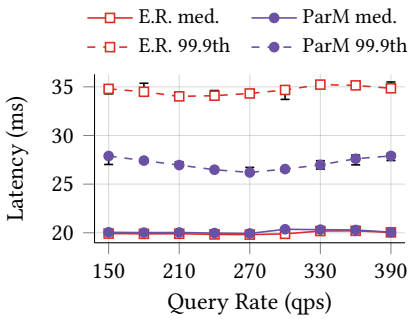


Figure 13. ParM and Equal-Resources (E.R.) with light inference multitenancy.

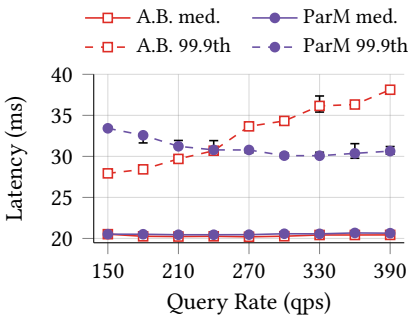


Figure 14. Latencies of ParM and using approximate backup models (A.B.).

only 1.15× faster on the GPU cluster. Thus, an approximate model designed for one hardware setup may not provide benefits on other hardware, limiting deployment flexibility.

Finally, this approach uses 2× network bandwidth by replicating queries. This can be problematic, as limited bandwidth has been shown to hinder prediction-serving [25, 38].

Some of the limitations of this approach may be mitigated by reactively issuing redundant queries to approximate backup models after a timeout. However, like other reactive approaches described in §2.2, doing so reduces the potential for such an approach to reduce tail latency.

ParM does not have the drawbacks described above. As described in §4, ParM's parity models have the same average runtime as deployed models, and ParM encodes k queries into one parity query prior to dispatching to a parity model. The $\frac{m}{k}$ parity models therefore receive $\frac{1}{k}$ the query rate of the m deployed models, and thus naturally keep pace. This reduced query rate also means that ParM adds only minor network bandwidth overhead. Further, by using the same architecture for parity models as is used for deployed models, ParM does not face hardware-related deployment issues.

7 Discussion

Training time. We find that the time to train a parity model can be 3-12× longer than that of a deployed model. Reducing training time was not a goal of this work. We next describe ways in which training time may potentially be reduced.

As described in §4.4, the training procedure for a parity model draws k samples from the deployed model's training dataset to form a single training sample for the parity model. Thus, the number of possible combinations of k samples that could be used grows exponentially with k. For example, training a parity model using a deployed model dataset with 1000 samples would lead to an effective dataset size of 1000^k. For large deployed model datasets, the effective parity model dataset is too large to train on every possible combination. We currently randomly sample combinations of k queries from the deployed model dataset without keeping track of which combinations have been used. As was shown in §5, even this simple approach enables accurate training of parity models. A more principled approach to sampling from the deployed model's dataset may help reduce training time and improve accuracy, especially for massive datasets and those with imbalance in the number of samples of a particular type (e.g., fewer examples of fish than dogs).

Throughput. Like any redundancy-based approach, the achievable throughput when using coded-computation is lower than it could be if one used all resources for serving copies of a deployed model. Specifically, as ParM uses $\frac{1}{k}$ of all model instances for parity models, ParM's maximum achievable throughput is $(1 - \frac{1}{k})$-times that of an approach which uses no redundancy. However, as shown in §6, reserving some resources to be used for redundancy aids in reducing tail latency. ParM enables one to span this tradeoff between tail latency and throughput by changing parameter k.

Concurrent unavailabilities. ParM can accommodate concurrent unavailabilities by using decoders parameterized with $r > 1$. In this case, r separate parity models can be trained to produce different transformations of a parity query. For example, consider having $k = 2, r = 2$, queries X_1 and X_2, and parity $P = (X_1 + X_2)$. One parity model can be trained to transform P into $\mathcal{F}(X_1) + \mathcal{F}(X_2)$, while the second can be trained to transform P into $\mathcal{F}(X_1) + 2\mathcal{F}(X_2)$. The decoder reconstructs the initial k predictions using any k out of the $(k + r)$ predictions from deployed models and parity models.

Parity model design space. In this work, we have chosen to keep encoders and decoders simple and to specialize parity models so as to reduce the additional computation introduced on prediction serving system frontends. However, as described in §4 and shown empirically in §5.2.3 using an image-classification-task specific encoder, ParM's framework opens a rich design space for encoders, decoders, and parity models. Navigating this design space by co-designing these components may yield interesting opportunities for improving the accuracy of reconstructions.

Applicability to other workloads. We have showcased the use of parity models on image classification, speech recognition, and object localization tasks. However, the core idea of learning-based coded-computation has the potential to be applied more broadly. For example, parity models may potentially be applicable to sequence-to-sequence models, such as those for translation, though further research is necessary to accommodate the use of parity models to these tasks.

8 Related Work

Mitigating slowdowns. Many approaches target specific causes of slowdown. Examples of such techniques include configuration selection [17, 58, 80, 90], isolation [33, 44, 61, 88], replica selection [35, 79], predicting slowdowns [89], and autoscaling [24, 34]. As these techniques apply only to specific slowdowns, they are unable to mitigate all slowdowns. In contrast, ParM is agnostic to the cause of slowdown.

Many techniques mitigate slowdowns in *training* [37, 41, 69]. These techniques exploit iterative computations specific to training and are thus inapplicable to inference.

Accuracy-latency tradeoff. A number of systems trade accuracy for lower latency [83, 95]. This enables handling query rate variation efficiently, but may degrade accuracy. In contrast, ParM does not proactively degrade accuracy; any inaccuracy due to ParM is incurred only when a prediction experiences slowdown or failure.

Algorithmic techniques like cascades [81, 84] and anytime neural networks [42] enable "early exit" during inference for queries that are easier to complete or those taking longer than a predefined deadline. These techniques are only applicable to reducing the latency of inference, and thus will not help mitigate tail latency induced by other sources, like network congestion or failure. In contrast, ParM alleviates slowdowns and failures regardless of their cause.

High performance inference. Many techniques improve the average latency and throughput of inference [10, 22, 45, 46, 55, 60]. These techniques are complementary to ParM, which is designed for mitigating slowdowns and failures.

Coded-computation. As described in §2.3, most existing coded-computation techniques support only rudimentary computations. A recent class of codes [78, 93] supports polynomial computations, but requires as many or more resources than replication-based approaches. Another approach [28] performs coded-computation over the linear operations of neural networks and decodes before each non-linear operation. This requires splitting the operations of a model onto multiple servers and many decoding steps, which increases latency even when predictions are not slow or failed. In contrast, ParM uses 2-4× less resource overhead than replication and does not require splitting neural networks onto separate servers or induce latency when predictions are available.

Learning error correcting codes. Multiple recent works have explored learning error correcting codes for communication [20, 49, 63]. The goal of these works is to recover data units transmitted over a noisy channel. In contrast, our goal is to recover the outputs of computation over data units. To the best of our knowledge, we present the first approach that leverages learning for coded-computation.

9 Conclusion

We present a fundamentally new, learning-based approach for enabling the use of ideas from erasure coding to impart low-latency, resource-efficient resilience to slowdowns and failures in prediction serving systems. Through judicious use of learning, parity models overcome the limitations of existing coded-computation approaches and enable the use of simple, fast encoders and decoders to reconstruct unavailable predictions for a variety of neural network inference tasks. We have built ParM, a prediction serving system that makes use of parity models, and shown the ability of ParM to reduce tail latency while maintaining high overall prediction accuracy in the face of load imbalance. These results suggest that our approach may open new doors for enabling the use of erasure-coded resilience for a broader class of workloads.

Acknowledgements

We thank our shepherd, Kai Shen, for providing valuable feedback, and Anuj Kalia and Greg Ganger for feedback on earlier versions of this paper. This work was funded in part by an NSF Graduate Research Fellowship (DGE-1745016 and DGE-1252522), in part by National Science Foundation grants CNS-1850483 and CNS-1838733, in part by Amazon Web Services, and in part by the Office of the Vice Chancellor for Research and Graduate Education at the University of Wisconsin, Madison with funding from the Wisconsin Alumni Research Foundation.

References

[1] Amazon Alexa. https://developer.amazon.com/alexa. Last accessed 01 September 2019.
[2] Amazon EC2 C5 Instances. https://aws.amazon.com/ec2/instance-types/c5/. Last accessed 01 September 2019.
[3] Azure Machine Learning Studio. https://azure.microsoft.com/en-us/services/machine-learning-studio/. Last accessed 01 September 2019.
[4] Google Cloud AI. https://cloud.google.com/products/machine-learning/. Last accessed 01 September 2019.
[5] Google lens: real-time answers to questions about the world around you. https://bit.ly/2MHAOLq. Last accessed 01 September 2019.
[6] HDFS RAID. http://www.slideshare.net/ydn/hdfs-raid-facebook. Last accessed 01 September 2019.
[7] iOS Siri. https://www.apple.com/ios/siri/. Last accessed 01 September 2019.
[8] Machine Learning on AWS. https://aws.amazon.com/machine-learning/. Last accessed 01 September 2019.
[9] Model Server for Apache MXNet. https://github.com/awslabs/mxnet-model-server. Last accessed 01 September 2019.

[10] NVIDIA TensorRT. https://developer.nvidia.com/tensorrt. Last accessed 01 September 2019.

[11] OpenCV. https://opencv.org/. Last accessed 01 September 2019.

[12] PyTorch. https://pytorch.org/. Last accessed 01 September 2019.

[13] Speculative Execution in Hadoop MapReduce. https://data-flair.training/blogs/speculative-execution-in-hadoop-mapreduce/. Last accessed 01 September 2019.

[14] Asirra: A CAPTCHA That Exploits Interest-aligned Manual Image Categorization. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 07)* (2007).

[15] Agarwal, D., Long, B., Traupman, J., Xin, D., and Zhang, L. LASER: A Scalable Response Prediction Platform for Online Advertising. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining (WSDM 14)* (2014).

[16] Alex Krizhevsky and Vinod Nair and Geoffrey Hinton. The CIFAR-10 and CIFAR-100 Datasets. https://www.cs.toronto.edu/~kriz/cifar.html.

[17] Alipourfard, O., Liu, H. H., Chen, J., Venkataraman, S., Yu, M., and Zhang, M. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (2017).

[18] Ananthanarayanan, G., Ghodsi, A., Shenker, S., and Stoica, I. Effective Straggler Mitigation: Attack of the Clones. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (2013).

[19] Ananthanarayanan, G., Kandula, S., Greenberg, A. G., Stoica, I., Lu, Y., Saha, B., and Harris, E. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)* (2010).

[20] Aoudia, F. A., and Hoydis, J. End-to-End Learning of Communications Systems Without a Channel Model. *arXiv preprint arXiv:1804.02276* (2018).

[21] Baylor, D., Breck, E., Cheng, H.-T., Fiedel, N., Foo, C. Y., Haque, Z., Haykal, S., Ispir, M., Jain, V., Koc, L., et al. TFX: A Tensorflow-Based Production-Scale Machine Learning Platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 17)* (2017).

[22] Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*.

[23] Chung, E., Fowers, J., Ovtcharov, K., Papamichael, M., Caulfield, A., Massengill, T., Liu, M., Lo, D., Alkalay, S., Haselman, M., et al. Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. *IEEE Micro 38*, 2 (2018), 8–20.

[24] Crankshaw, D., Sela, G.-E., Zumar, C., Mo, X., Gonzalez, J. E., Stoica, I., and Tumanov, A. InferLine: ML Inference Pipeline Composition Framework. *arXiv preprint arXiv:1812.01776* (2018).

[25] Crankshaw, D., Wang, X., Zhou, G., Franklin, M. J., Gonzalez, J. E., and Stoica, I. Clipper: A Low-Latency Online Prediction Serving System. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (2017).

[26] Dean, J., and Barroso, L. A. The Tail at Scale. *Communications of the ACM 56*, 2 (2013), 74–80.

[27] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *International Conference on Learning Representations (ICLR 15)* (2015).

[28] Dutta, S., Bai, Z., Jeong, H., Low, T. M., and Grover, P. A Unified Coded Deep Neural Network Training Strategy Based on Generalized Polydot Codes for Matrix Multiplication. In *Proceedings of the 2018 IEEE International Symposium on Information Theory (ISIT 18)* (2018).

[29] Dutta, S., Cadambe, V., and Grover, P. Short-dot: Computing Large Linear Transforms Distributedly Using Coded Short Dot Products. In *Advances In Neural Information Processing Systems (NIPS 16)* (2016).

[30] Dutta, S., Cadambe, V., and Grover, P. Coded Convolution for Parallel and Distributed Computing Within a Deadline. In *Proceedings of the 2017 IEEE International Symposium on Information Theory (ISIT 17)* (2017).

[31] Gardner, K., Zbarsky, S., Doroudi, S., Harchol-Balter, M., and Hyytia, E. Reducing Latency via Redundant Requests: Exact Analysis. *ACM SIGMETRICS Performance Evaluation Review 43*, 1 (2015), 347–360.

[32] Glorot, X., and Bengio, Y. Understanding the Difficulty of Training Deep Feedforward Neural Networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS 10)* (2010).

[33] Grosvenor, M. P., Schwarzkopf, M., Gog, I., Watson, R. N. M., Moore, A. W., Hand, S., and Crowcroft, J. Queues Don't Matter When You Can JUMP Them! In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (2015).

[34] Gujarati, A., Elnikety, S., He, Y., McKinley, K. S., and Brandenburg, B. B. Swayam: Distributed Autoscaling to Meet SLAs of Machine Learning Inference Services with Resource Efficiency. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference (Middleware 17)* (2017).

[35] Hao, M., Li, H., Tong, M. H., Pakha, C., Suminto, R. O., Stuardo, C. A., Chien, A. A., and Gunawi, H. S. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP 17)* (2017).

[36] Harchol-Balter, M. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, 2013.

[37] Harlap, A., Cui, H., Dai, W., Wei, J., Ganger, G. R., Gibbons, P. B., Gibson, G. A., and Xing, E. P. Addressing the Straggler Problem for Iterative Convergent Parallel ML. In *Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC 16)* (2016).

[38] Hauswald, J., Kang, Y., Laurenzano, M. A., Chen, Q., Li, C., Mudge, T., Dreslinski, R. G., Mars, J., and Tang, L. DjiNN and Tonic: DNN as a Service and Its Implications for Future Warehouse Scale Computers. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA 15)* (2015).

[39] Hazelwood, K., Bird, S., Brooks, D., Chintala, S., Diril, U., Dzhulgakov, D., Fawzy, M., Jia, B., Jia, Y., Kalro, A., et al. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA 18)* (2018).

[40] He, K., Zhang, X., Ren, S., and Sun, J. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR 16)* (2016).

[41] Ho, Q., Cipar, J., Cui, H., Lee, S., Kim, J. K., Gibbons, P. B., Gibson, G. A., Ganger, G., and Xing, E. P. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In *Advances in Neural Information Processing Systems (NIPS 13)* (2013).

[42] Hu, H., Dey, D., Bagnell, J. A., and Hebert, M. Learning Anytime Predictions in Neural Networks via Adaptive Loss Balancing. *arXiv preprint arXiv:1708.06832* (2018).

[43] Huang, C., Simitci, H., Xu, Y., Ogus, A., Calder, B., Gopalan, P., Li, J., and Yekhanin, S. Erasure Coding in Windows Azure Storage. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)* (2012).

[44] Iorgulescu, C., Azimi, R., Kwon, Y., Elnikety, S., Syamala, M., Narasayya, V., Herodotou, H., Tomita, P., Chen, A., Zhang, J., and Wang, J. PerfIso: Performance Isolation for Commercial Latency-Sensitive Services. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (2018).

[45] Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., and Kalenichenko, D. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. *arXiv preprint*

arXiv:1712.05877 (2017).

[46] Jiang, A. H., Wong, D. L.-K., Canel, C., Tang, L., Misra, I., Kaminsky, M., Kozuch, M. A., Pillai, P., Andersen, D. G., and Ganger, G. R. Mainstream: Dynamic Stem-Sharing for Multi-Tenant Video Processing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (2018).

[47] Joshi, G., Liu, Y., and Soljanin, E. On the Delay-Storage Trade-Off in Content Download From Coded Distributed Storage Systems. *IEEE JSAC*, 5 (2014), 989–997.

[48] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA 17)* (2017).

[49] Kim, H., Jiang, Y., Rana, R., Kannan, S., Oh, S., and Viswanath, P. Communication Algorithms via Deep Learning. In *International Conference on Learning Representations (ICLR 18)* (2018).

[50] Kosaian, J., Rashmi, K. V., and Venkataraman, S. Learning a Code: Machine Learning for Approximate Non-Linear Coded Computation. *arXiv preprint arXiv:1806.01259* (2018).

[51] Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems (NIPS 12)* (2012).

[52] LeCun, Y. The MNIST database of handwritten digits. `http://yann.lecun.com/exdb/mnist/`.

[53] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based Learning Applied to Document Recognition. *Proceedings of the IEEE 86*, 11 (1998), 2278–2324.

[54] Lee, K., Lam, M., Pedarsani, R., Papailiopoulos, D., and Ramchandran, K. Speeding Up Distributed Machine Learning Using Codes. *IEEE Transactions on Information Theory* (July 2018).

[55] Lee, Y., Scolari, A., Chun, B.-G., Santambrogio, M. D., Weimer, M., and Interlandi, M. PRETZEL: Opening the Black Box of Machine Learning Prediction Serving Systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (2018).

[56] Lee, Y., Scolari, A., Interlandi, M., Weimer, M., and Chun, B.-G. Towards High-Performance Prediction Serving Systems. *NIPS ML Systems Workshop* (2017).

[57] Li, S., Maddah-Ali, M. A., and Avestimehr, A. S. A Unified Coding Framework for Distributed Computing With Straggling Servers. In *2016 IEEE Globecom Workshops (GC Wkshps)* (2016).

[58] Li, Z. L., Liang, C.-J. M., He, W., Zhu, L., Dai, W., Jiang, J., and Sun, G. Metis: Robustly Tuning Tail Latencies of Cloud Systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (2018).

[59] Liang, G., and Kozat, U. C. FAST CLOUD: Pushing the Envelope on Delay Performance of Cloud Storage with Coding. *arXiv:1301.1294* (Jan. 2013).

[60] Liu, Y., Wang, Y., Yu, R., Li, M., Sharma, V., and Wang, Y. Optimizing CNN Model Inference on CPUs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (2019).

[61] Mace, J., Bodik, P., Musuvathi, M., Fonseca, R., and Varadarajan, K. 2DFQ: Two-Dimensional Fair Queuing for Multi-Tenant Cloud Services. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM 16)* (2016).

[62] Mallick, A., Chaudhari, M., and Joshi, G. Rateless Codes for Near-Perfect Load Balancing in Distributed Matrix-Vector Multiplication. *arXiv preprint arXiv:1804.10331* (2018).

[63] Nachmani, E., Marciano, E., Lugosch, L., Gross, W. J., Burshtein, D., and Be'ery, Y. Deep Learning Methods for Improved Decoding of Linear Codes. *IEEE Journal of Selected Topics in Signal Processing 12*, 1 (2018), 119–131.

[64] Olston, C., Fiedel, N., Gorovoy, K., Harmsen, J., Lao, L., Li, F., Rajashekhar, V., Ramesh, S., and Soyke, J. TensorFlow-Serving: Flexible, High-Performance ML Serving. *NIPS ML Systems Workshop* (2017).

[65] Park, J., Naumov, M., Basu, P., Deng, S., Kalaiah, A., Khudia, D., Law, J., Malani, P., Malevich, A., Nadathur, S., et al. Deep Learning Inference in Facebook Data Centers: Characterization, Performance Optimizations and Hardware Implications. *arXiv preprint arXiv:1811.09886* (2018).

[66] Patterson, D. A., Gibson, G., and Katz, R. H. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 88)* (1988).

[67] Rashmi, K. V., Chowdhury, M., Kosaian, J., Stoica, I., and Ramchandran, K. EC-Cache: Load-Balanced, Low-Latency Cluster Caching with Online Erasure Coding. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016).

[68] Rashmi, K. V., Shah, N. B., Gu, D., Kuang, H., Borthakur, D., and Ramchandran, K. A Hitchhiker's Guide to Fast and Efficient Data Reconstruction in Erasure-Coded Data Centers. In *Proceedings of the 2014 ACM SIGCOMM Conference (SIGCOMM 14)* (2014).

[69] Recht, B., Re, C., Wright, S., and Niu, F. Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *Advances in Neural Information Processing Systems (NIPS 11)* (2011).

[70] Reed, I. S., and Solomon, G. Polynomial Codes Over Certain Finite Fields. *Journal of the society for industrial and applied mathematics 8*, 2 (1960), 300–304.

[71] Reisizadeh, A., Prakash, S., Pedarsani, R., and Avestimehr, S. Coded Computation Over Heterogeneous Clusters. In *Proceedings of the 2017 IEEE International Symposium on Information Theory (ISIT 17)* (2017).

[72] Richardson, T., and Urbanke, R. *Modern Coding Theory*. Cambridge University Press, 2008.

[73] Rizzo, L. Effective Erasure Codes for Reliable Computer Communication Protocols. *ACM SIGCOMM Computer Communication Review 27*, 2 (1997), 24–36.

[74] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV) 115*, 3 (2015), 211–252.

[75] Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. MobilenetV2: Inverted Residuals and Linear Bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR 18)* (2018).

[76] Shah, N. B., Lee, K., and Ramchandran, K. When do Redundant Requests Reduce Latency? *IEEE Transactions on Communications 64*, 2 (2016), 715–722.

[77] Simonyan, K., and Zisserman, A. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *International Conference on Learning Representations (ICLR 15)* (2015).

[78] So, J., Guler, B., Avestimehr, A. S., and Mohassel, P. CodedPrivateML: A Fast and Privacy-Preserving Framework for Distributed Machine Learning. *arXiv preprint arXiv:1902.00641* (2019).

[79] Suresh, L., Canini, M., Schmid, S., and Feldmann, A. C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (2015).

[80] Venkataraman, S., Yang, Z., Franklin, M., Recht, B., and Stoica, I. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (2016).

[81] Viola, P., and Jones, M. J. Robust Real-Time Face Detection. *International Journal of Computer Vision 57*, 2 (2004), 137–154.

[82] Wang, S., Liu, J., and Shroff, N. Coded Sparse Matrix Multiplication. In *Proceedings of the International Conference on Machine Learning (ICML 18)* (2018).

[83] Wang, W., Gao, J., Zhang, M., Wang, S., Chen, G., Ng, T. K., Ooi, B. C., Shao, J., and Reyad, M. Rafiki: Machine Learning as an Analytics Service System. *Proceedings of the VLDB Endowment 12*, 2 (2018), 128–140.

[84] Wang, X., Luo, Y., Crankshaw, D., Tumanov, A., Yu, F., and Gonzalez, J. E. IDK Cascades: Fast Deep Learning by Learning not to Overthink. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI 18)* (2018).

[85] Warden, P. Speech commands: A Dataset for Limited-Vocabulary Speech Recognition. *arXiv preprint arXiv:1804.03209* (2018).

[86] Welinder, P., Branson, S., Mita, T., Wah, C., Schroff, F., Belongie, S., and Perona, P. Caltech-UCSD Birds 200. Tech. Rep. CNS-TR-2010-001, California Institute of Technology, 2010.

[87] Xiao, H., Rasul, K., and Vollgraf, R. Fashion-Mnist: A Novel Image Dataset for Benchmarking Machine Learning Algorithms. *arXiv preprint arXiv:1708.07747* (2017).

[88] Xu, Y., Musgrave, Z., Noble, B., and Bailey, M. Bobtail: Avoiding Long Tails in the Cloud. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (2013).

[89] Yadwadkar, N. J., Ananthanarayanan, G., and Katz, R. Wrangler: Predictable and Faster Jobs using Fewer Resources. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC 14)* (2014).

[90] Yadwadkar, N. J., Hariharan, B., Gonzalez, J. E., Smith, B., and Katz, R. H. Selecting the Best VM Across Multiple Public Clouds: A Data-Driven Performance Modeling Approach. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC 17)* (2017).

[91] Yan, S., Li, H., Hao, M., Tong, M. H., Sundararaman, S., Chien, A. A., and Gunawi, H. S. Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. In *15th USENIX Conference on File and Storage Technologies (FAST 17)* (2017).

[92] Yu, Q., Maddah-Ali, M., and Avestimehr, S. Polynomial Codes: An Optimal Design for High-Dimensional Coded Matrix Multiplication. In *Advances in Neural Information Processing Systems (NIPS 17)* (2017).

[93] Yu, Q., Raviv, N., So, J., and Avestimehr, A. S. Lagrange Coded Computing: Optimal Design for Resiliency, Security and Privacy. In *Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics (AISTATS 19)* (2019).

[94] Zaharia, M., Konwinski, A., Joseph, A. D., Katz, R., and Stoica, I. Improving MapReduce Performance in Heterogeneous Environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI 08)* (2008).

[95] Zhang, H., Ananthanarayanan, G., Bodik, P., Philipose, M., Bahl, P., and Freedman, M. J. Live Video Analytics at Scale with Approximation and Delay-Tolerance. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (2017).

[96] Zhang, M., Rajbhandari, S., Wang, W., and He, Y. DeepCPU: Serving RNN-based Deep Learning Models 10x Faster. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (2018).

[97] Zoph, B., and Le, Q. V. Neural Architecture Search with Reinforcement Learning. *arXiv preprint arXiv:1611.01578* (2016).